Proceedings of DETC'01
ASME 2001 Design Engineering Technical Conferences and Information in Engineering Conference
Pittsburgh, Pennsylvania, September 9-12, 2001

EUC-03

TIMING STUDIES OF REAL-TIME LINUX FOR CONTROL

Frederick M. Proctor and William P. Shackleford
Control Systems Group
National Institute of Standards and Technology
100 Bureau Drive, Stop 8230
Gaithersburg, MD 20899-8230 USA

ABSTRACT

Linux is being used increasingly for real-time control of industrial equipment. Versions of Linux adapted to support deterministic task execution are freely available. The resolution of task timing is much higher than for typical user-level processes, on the order of tens of microseconds. At this level, timing jitter due to hardware effects is visible.

This paper describes Linux and real-time Linux, its application for industrial control, and shows the results of some timing studies. Various timing techniques are presented and their advantages and disadvantages are discussed. A method for improving timing is presented.

INTRODUCTION

Linux is a clone of the Unix operating system written by Linus Torvalds, a student at the University of Helsinki in Finland, with assistance from programmers across the Internet [1]. The project was started in 1991 and version 1.0 was released in 1994. It includes features commonly expected from modern operating systems, including multiprocessing, multitasking, virtual memory, shared libraries, demand loading, memory management, and TCP/IP networking.

Developed under the GNU General Public License [2], the source code for Linux is freely available to everyone. It is portable to most general-purpose 32- or 64-bit architectures as long as they have a paged memory management unit (PMMU) and a port of the GNU C compiler. First developed for 32-bit Intel X86-based PCs, Linux has been ported to the Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68000, PowerPC, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64 and DEC VAX*.

The free nature of the Linux source code and its availability on a wide range of processor architectures has made it popular in the embedded systems community. Embedded systems typically have a small hardware footprint, need to minimize use of computing and power resources, and operate in environments that do not tolerate rotating storage media.

Many embedded systems tolerate some variation in task execution times and do not require a guarantee that tasks will complete before a deadline. These are categorized as *soft real-time* systems, where a statistical distribution for response time is acceptable. *Hard real-time systems*, in contrast, require that some tasks must be completed within a certain time interval or incorrect operation will occur [3]. As a variant of Unix, Linux is optimized for best average response time. However, Linux is not a real-time operating system: it does not guarantee that tasks will execute before a certain deadline.

Several groups have made hard real-time modifications to Linux. The New Mexico Institute of Mining and Technology developed Real-Time Linux (NMT RTL) [4] that runs on x86, PowerPC and Alpha platforms. The Department of Aerospace Engineering of the Polytechnic Institute of Milan developed the Real Time Application Interface (RTAI) [5], that runs on x86 and PowerQUICC processors. These versions of real-time Linux are available free as patches to the Linux source code, or through commercial vendors who provide documentation and support.

Real-time system development requires selecting a real-time operating system that provides scheduling guarantees, and designing the software to ensure that deadlines are met. Techniques such as rate-monotonic analysis (RMA) [6, 7, 8] are commonly used to derive task scheduling based on task worst-case execution time (WCET). However, worst-case execution time is difficult to obtain analytically for tasks running on general-purpose microprocessors like the Intel Pentium, due to unpredictable effects from external sources like interrupts and bus blocking, and internal sources like caches and pipelines [9, 10].

Variation in task execution time confounds efforts to measure WCET for application tasks, but it also affects system tasks like the scheduler and compounds the problem. Typically real-time tasks are scheduled at a regular period, and initiated when a hardware timer generates an interrupt that wakes up the scheduler. While the timer interrupt may occur with clock-like regularity, the scheduler may run quickly or slowly depending

upon the many effects listed previously. The subsequent initiation of the application task will vary accordingly. This effect is called *scheduling jitter*.

Because of the unpredictability in task execution time and scheduling on general purpose microprocessors, real-time systems usually run on chip architectures such as digital signal processors (DSPs) without them. To make up for the lack of performance-enhancing features such as caches and pipelines, DSPs provide features such as fast multiply operations or instructions that work on parallel data sets.

Despite the problems inherent in pipelined cached processors like the Pentium, these general-purpose chips are an attractive target for real-time system developers due to their use in the ubiquitous PC-compatible computer. PC-compatibles also include many useful features such as mass storage, video output, Ethernet connectivity, and support for a wide variety of ISA- and PCI-bus input/output cards. Rather than forgo the utilitarian PC-compatible altogether, it is worthwhile to ask the questions, "How bad is the problem? And what can be done about it?"

MEASURING SCHEDULING JITTER

One way to measure scheduling jitter is to run a simple periodic task that logs time stamps of its invocation, and then analyze the logged times for variation. In our testing, a trivial periodic task was run by the RT Linux scheduler, attached to a timer interrupt from the computer's 8254 programmable interval timer (PIT) chip. This is known as "pure periodic mode" scheduling. The PIT is not reprogrammed further, and is not a source of appreciable timing jitter. The task executes the Pentium RDTSC instruction to read the Time Stamp Counter (TSC), a 64-bit integer internal to the Pentium that increments once each clock cycle [11]. The resolution of the counter is the reciprocal of the clock frequency, 2.5 nanoseconds for a 400-megahertz clock. All the TSC values are logged into RAM and then later saved to disk for analysis. The logged values should be exactly the interrupt period apart, but variations in the combined execution times of the scheduler and task code prior to the RDTSC instruction show up as deviations from the nominal period. For the purpose of estimating scheduling jitter, the effect of the task code prior to the RDTSC is attributed to the scheduler. This is a small amount of code (such as setting up stack variables) that would undoubtedly be present in all tasks and thus acceptable to attribute to the scheduling process.

Jitter can be estimated from the TSC log in one of two ways, each which can be physically interpreted as if the task were a square wave pulse generator. Figure 1 shows a hypothetical log of time stamps for a task that toggles an output high or low with each invocation, and which experiences a single task delay. In this figure, the task cycle nominally expected to occur at 2000 _sec actually occurs at 2200 _sec. The subsequent task cycle occurs on schedule at 2500 _sec. With the first method, the differences between adjacent TSC values are computed. In terms of the pulse generator, the largest TSC difference will generate the widest output crest (or trough), while the smallest difference will generate the narrowest. Jitter is then defined as the difference between the widest and narrowest pulse. However, as seen in Figure 1, a single late invocation of the task will be manifested as a long pulse followed by a short pulse, the second being shorter only because its predecessor was delayed. In this illustrative example, each invocation of the task toggles the output between high and low states. In the ideal case where there is a constant time interval between the scheduling interrupt and the task invocation, the pulses are equally wide. In the figure, the task cycle expected at 2000 _sec has been delayed due to a jitter effect. The pulse is correspondingly longer. The subsequent task invocation occurs on schedule at 2500 _sec, but the previous delay shortens the width of the its associated pulse. This effectively estimates jitter as double the scheduling delay. In the case of the pulse generator, however, jitter as a measure of the difference between longest and shortest pulse is certainly meaningful.

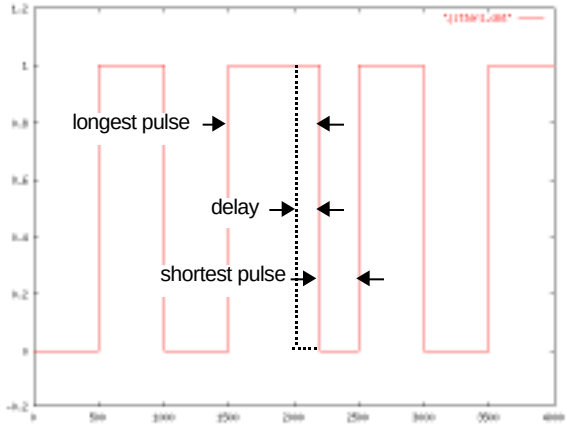

Figure 1. The effect of a single late task invocation on pulse widths, for a square wave pulse generator task running at a 500 _sec nominal period. The effect of the jitter penalty on adjacent pulses effectively doubles its contribution to overall scheduling jitter, as defined as the difference between longest and shortest intervals.

With the second method, the differences between each TSC value and its expected nominal value are computed. The expected nominal values are not known; these are the time stamps that would be expected if there were a constant time interval between the timer interrupt and the task invocation. The nominal values can be estimated as those that lie on the best-fit line to the actual TSC values. The difference between each TSC value and its nominal value is its deviation. Jitter is then defined as the difference between the maximum and minimum deviations. In terms of the pulse generator, jitter is approximately the extra width of the longest pulse. The subsequent shorter pulse is not included in the penalty since its task invocation was on schedule. Jitter measured in this way will be about half that in the first method.

COMPARISONS OF THE TWO METHODS

Figure 2 shows a plot of the differences between adjacent TSC values measured during a log of 10,000 points by a

periodic task running at 500 _sec. Note that the high values are mirrored by corresponding low values, due to the effect shown in Figure 1. The jitter is defined as the range between the highest (502.55 _sec) and lowest (496.89 _sec) values, or 5.66 _sec.

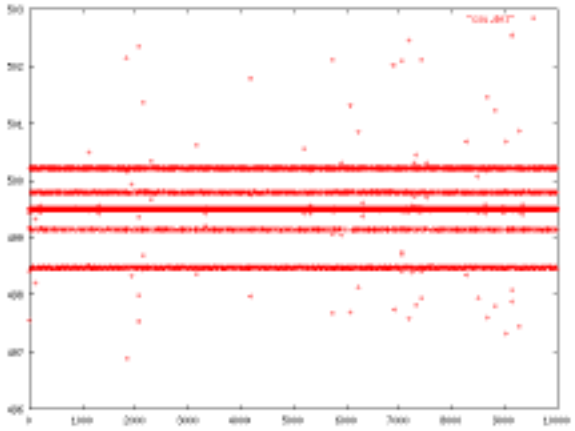

Figure 2. Cycle-to-cycle jitter plot. This shows the difference between adjacent values of the Pentium time-stamp counter, logged by a periodic task running at 500 _sec. 10,000 points were logged. With this method jitter is defined as the difference between highest (502.55 _sec) and lowest (496.89 _sec) values, or 5.66 _sec. Note the mirroring of points above and below the 500-_sec interval, due to the effect shown in Figure 1.

Figure 3 shows plot of the differences between TSC values and their nominal values from the best-fit line. The source data is exactly the same as for Figure 2; only the analysis of the logged data is different. The jitter is defined as the range between the highest (3.40 _sec) and lowest (-0.20 _sec) sample differences from the nominal, or 3.60 _sec. Note the absence of mirroring, that is, the high jitter points do not have negative counterparts. This effectively reduces the range and results in a jitter figure about half that of the first method.

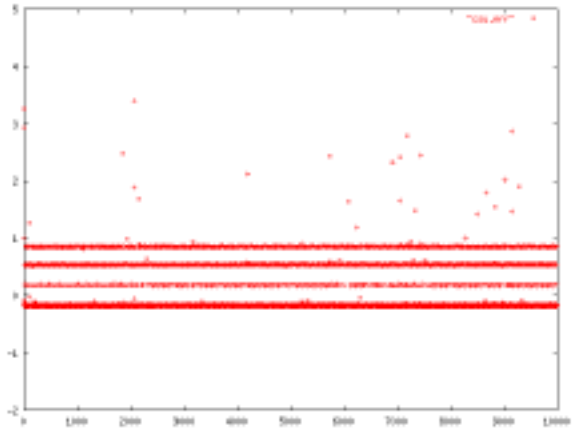

Figure 3. Time-base jitter plot. This shows the difference between actual values of the Pentium time-stamp counter and their nominal values expected from a constant time interval, logged by a periodic task running at 500 _sec. 10,000 points were logged. With this method jitter is defined as the difference between highest (3.40 _sec) and lowest (-0.20 _sec) values, or 3.60 _sec.

Either method is an acceptable way to calculate scheduling jitter. Which method is used will determine how the result is interpreted. If cycle-to-cycle variation is important, the first method may be more appropriate. If time-base variation is important, the second is more appropriate.

Time-base jitter plots can reveal trends that are hidden with cycle-to-cycle jitter plots. Analysis of the one-shot mode scheduler in an early version of RTL shows this. One-shot scheduling is useful when running several concurrent tasks with periods that are not multiples of a reasonable base period. The one-shot scheduler reprograms the 8254 PIT at the conclusion of each task, loading the time interval until the next scheduled task. Because of a bug in the early RTL one-shot mode scheduler, the task invocation drifted relative to a fixed period, and was periodically resynchronized. Cycle-to-cycle jitter plots will not show this in an obvious way. With time-base jitter plots, this trend is obvious.

Figures 4 and 5 show the cycle-to-cycle and time-base jitter plots of the TSC logging task run with one-shot scheduling. Figure 4 shows periodic high-jitter points due to resynchronization. Figure 5 shows the resynchronization and also the drift trend during the intervening cycles. Note that the vertical scales on these figures are much greater than for previous figures, so the _sec-level jitter is not visible.

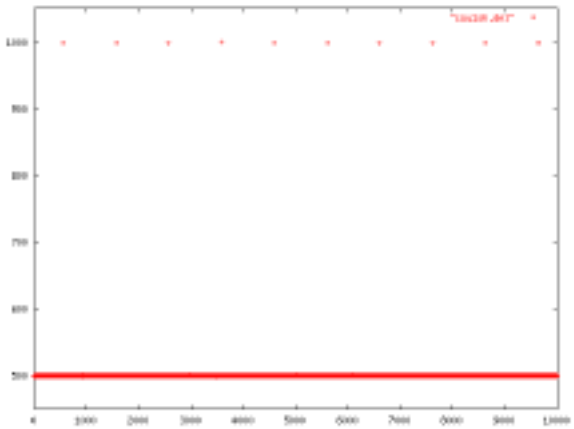

Figure 4. A cycle-to-cycle jitter plot of one-shot scheduling. The occasional high-jitter points are due to the resychronization of the scheduler to the time base, to compensate for drift.

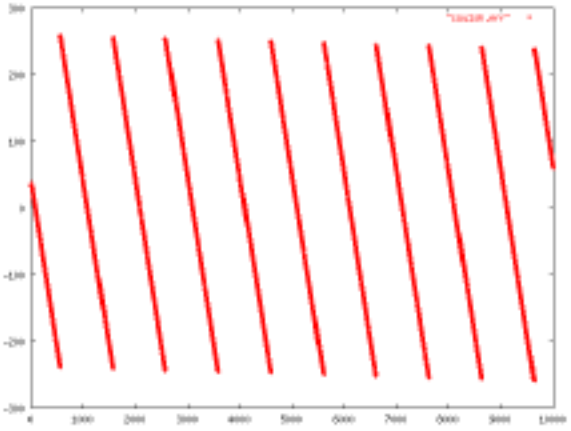

Figure 5. A time-base jitter plot of the same TSC data used for Figure 4. Scheduling drift in the interval between resynchronization is evident.

JITTER BANDS

Figure 3 showed pronounced bands about a third of a _sec apart, indicating a clustering of time stamp deviations from the nominal time base. Figure 6 shows a frequency histogram of the same source data as in Figure 3, with peaks associated with each band. The figure shows that most of the samples are clustered at the peak to the left, which lies about 0.2 _sec earlier than the zero point of the best fit. The other peaks occur about 0.4 _sec, 0.8 _sec, and 1 _sec later.

Frequency histograms for data logs at other time periods are shown in Figure 7. The time periods of the tasks for which the data was logged were 50, 100, 200, 300, 400, and 500 _sec from the top to the bottom, respectively. With the majority peaks aligned at the left, the locations of the subsequent peaks are clearly correlated. There are 8 peaks altogether. Analysis of the histogram data showed that these peaks occur at intervals -0.20, 0.04, 0.16, 0.34, 0.52, 0.70, and 0.83 _sec from the zero point of the best fit, or 0.24, 0.36, 0.54, 0.72, and 0.90 _sec after the majority peak.

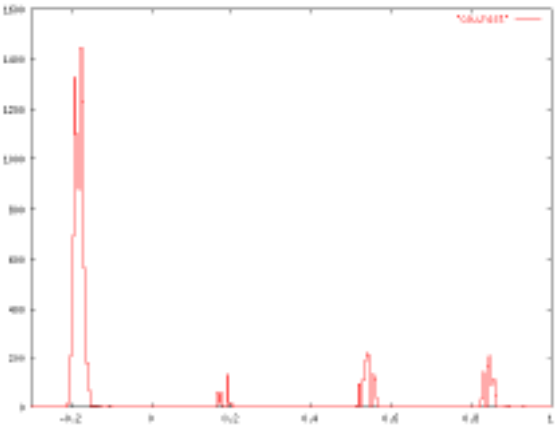

Figure 6. A frequency histogram of the data from Figure 3. The jitter is clustered into peaks 0.4 _sec, 0.8 _sec, and 1 _sec after the majority peak.

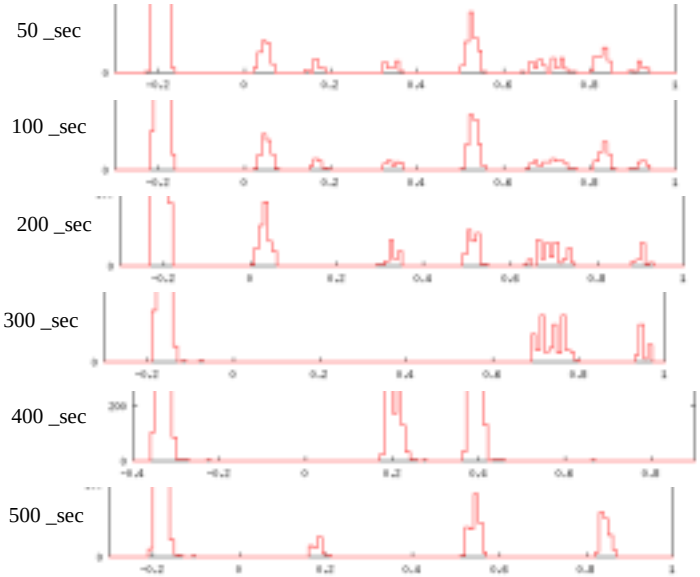

Figure 7. Plots of the frequency histograms for tasks running at 50, 100, 200, 300, 400, and 500 _sec from top to bottom, respectively. The plots have been aligned with the majority peaks, and show that the subsequent peaks associated with delayed invocation are correlated.

The regularity of the jitter peaks across different timing tests suggests a common origin to the late invocations of the time stamping task. Candidates include the scheduler and task cache access patterns, the effect on the cache from other tasks that execute during the overall logging interval, and the variation in instruction length between different branches of the scheduler code. Tracking down the precise sources would undoubtedly uncover some over which the programmer has control (such as interrupts and branch instruction length variation), and others that are endemic to the Pentium architecture itself and about which nothing can be done in software.

The time-base jitter plots in Figure 8 shows more clearly the effect of the cache. These plots show the first 100 points of the six timing tasks of period 50, 100, 200, 300, 400, and 500 _sec. Note that for each test, the first task invocations were the worst in terms of delay between scheduler interrupt and task invocation.

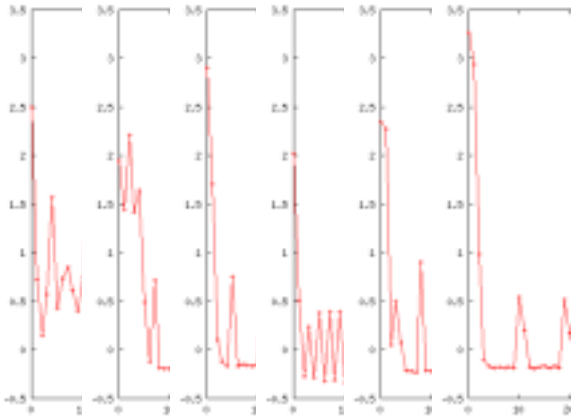

Figure 8. A close-up view of the first 100 points of the time-based jitter plots for the six timing tests. As in Figure 7, the tasks ran with nominal periods of 50, 100, 200, 300, 400, and 500 _sec, from left to right, respectively. Note the high first value in each plot, consistent with the cache loading penalties. For each full data set, the worst value was among the first.

REDUCING SCHEDULING JITTER

One method to reduce scheduling jitter, suggested by Tomasz Motylewski of the University of Basel, is for the task to delay its code by at least the maximum time-base scheduling jitter, beginning each cycle by polling the TSC repeatedly until the time for the desired period is reached. The amount of time spent looping will be less than the maximum scheduling jitter. For a 500-_sec period and 5-_sec maximum jitter, this is about 1% additional load. The results for this compensation are shown in Figure 10. The jitter for the compensated test is 0.098 _sec, while that for the uncompensated is 3.60 _sec. This is an improvement greater than a factor of 36.

This method is complicated by the possibilities of drift between the nominal period and the actual period programmed into the 8254 PIT. If the actual period is smaller, more time will be spent looping on the TSC as time progresses. If the actual period is larger, less time will be spent looping. This drift can be detected, since the time spent in the TSC loop should never exceed the maximum measured jitter nor be less than zero.

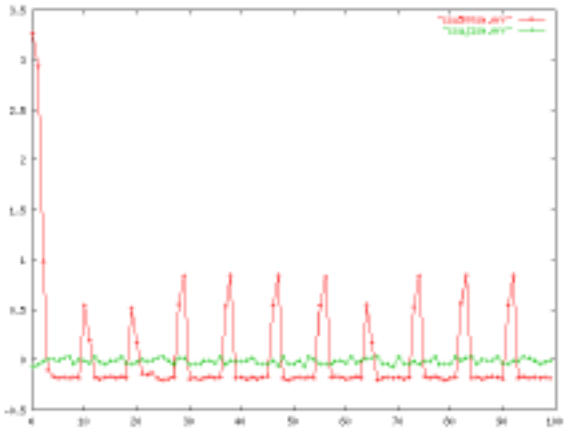

Figure 10. A comparison of the jitter for uncompensated scheduling, and compensating scheduling in which the task polls the TSC until the desired time is matched. The jitter is much lower (0.098 _s v. 3.60 _s) at the expense of about 1% processor time.

SUMMARY

Linux is being increasingly used for embedded and real-time applications. Real-time versions of Linux exist, and provide deterministic scheduling at the level of tens of microseconds. However, general-purpose microprocessors introduce scheduling jitter due to features such as caches and pipelines that vary instruction execution time. Two methods can be used to measure scheduling jitter, each which has its advantages. A technique was described that can reduce scheduling jitter by a significant factor, which makes the use of general-purpose microprocessors for real-time tasks a possibility.

REFERENCES

1. Linux Online, “What Is Linux?” Web resource: http://www.linux.org
2. The Free Software Foundation, GNU General Public License. Web resource: http://www.gnu.org/copyleft/gpl.html
3. Manacher, G. K., “Production and Stabilization of Real-Time Task Schedules,” Journal of the ACM, Vol. 14, No. 3, July 1967, pp. 439-465.
4. Real-Time Linux. Web resource: http://www.rtlinux.org
5. Real-Time Application Interface. Web resource: http://www.aero.polimi.it/projects/rtai
6. Liu, C. L. & Layland, J. W. “Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment.” Journal of the Association for Computing Machinery 20, 1 (January 1973): 40-61.
7. Sha, Klein & Goodenough, “Rate Monotonic Analysis for Real-Time Systems,” *Foundations of Real-Time Computing: Scheduling and Resource Management,* pp. 129-155. Boston, MA: Kluwer Academic Publishers. 1991.

8. Klein, M.H., et al. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Publishers, 1993.
9. Stappert, Friedhelm. "Predicting Pipelining and Caching Behavior of Hard Real-Time Programs," Proceedings of the Ninth Euromicro Workshop on Real-Time Systems, pp. 80-86, 1997.
10. Zhang, Lichen. "Worst Case Timing Analysis For Real-Time Programs," Proceedings of the 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing - Networking the Pacific Rim, Volume 2, pp. 960 -963, 1997.
11. Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Order Number 243191, 1999. Web reference: http://developer.intel.com/design/pentiumii/manuals/243191.htm

[*] Commercial equipment and materials are identified in order to specify adequately certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are the best available for the purpose.